3. Developing a Verifiable System

From **"**Verification, Validation, and Evaluation of Expert Systems, Volume I"

This chapter delineates how VV&E should be incorporated into the expert system lifecycle. Although some ideas may be used for revising and/or reengineering existing systems, this chapter is aimed mainly at designing new systems and ensuring that enough VV&E operations are done during the lifecycle so that these systems are verifiable. Included in this process are decisions that should be made during system specification and verification/validation during stepwise development of an expert system.

Introduction

The proposed lifecycle for the development of expert systems is a compilation of concepts taken from many sources including lifecycle, cleanroom, ect. The compiled system was organized and enhanced based on the experience of its developers to generate a basis for the development of "verifiable" systems. Even though the system allows for some flexibility in the degree of application of each of the system's components, the general outline has to be followed rigorously in order to achieve the objective outlined above.

The Concept: Figure 3.1. outlines the general concept for the development of a verifiable system. It includes the following stages:

Specification: This step is indispensable in the VV&E process.

Stepwise Development Process: This is one of the methods for the development process; other software development methods can be used as long as they include enough structure and verification steps.

Design (1): Start by designing the main parts of the expert system.

Verify (1): Verify that the design complies with the specification.

Implement (1): Implement (code) the first increment.

Verify (1): Verify that the implemented code complies with the design.

Design - Verify - Implement - Verify (2 to n): Loop through the entire process for the 2nd, 3rd, ... nth level until the entire system is complete.

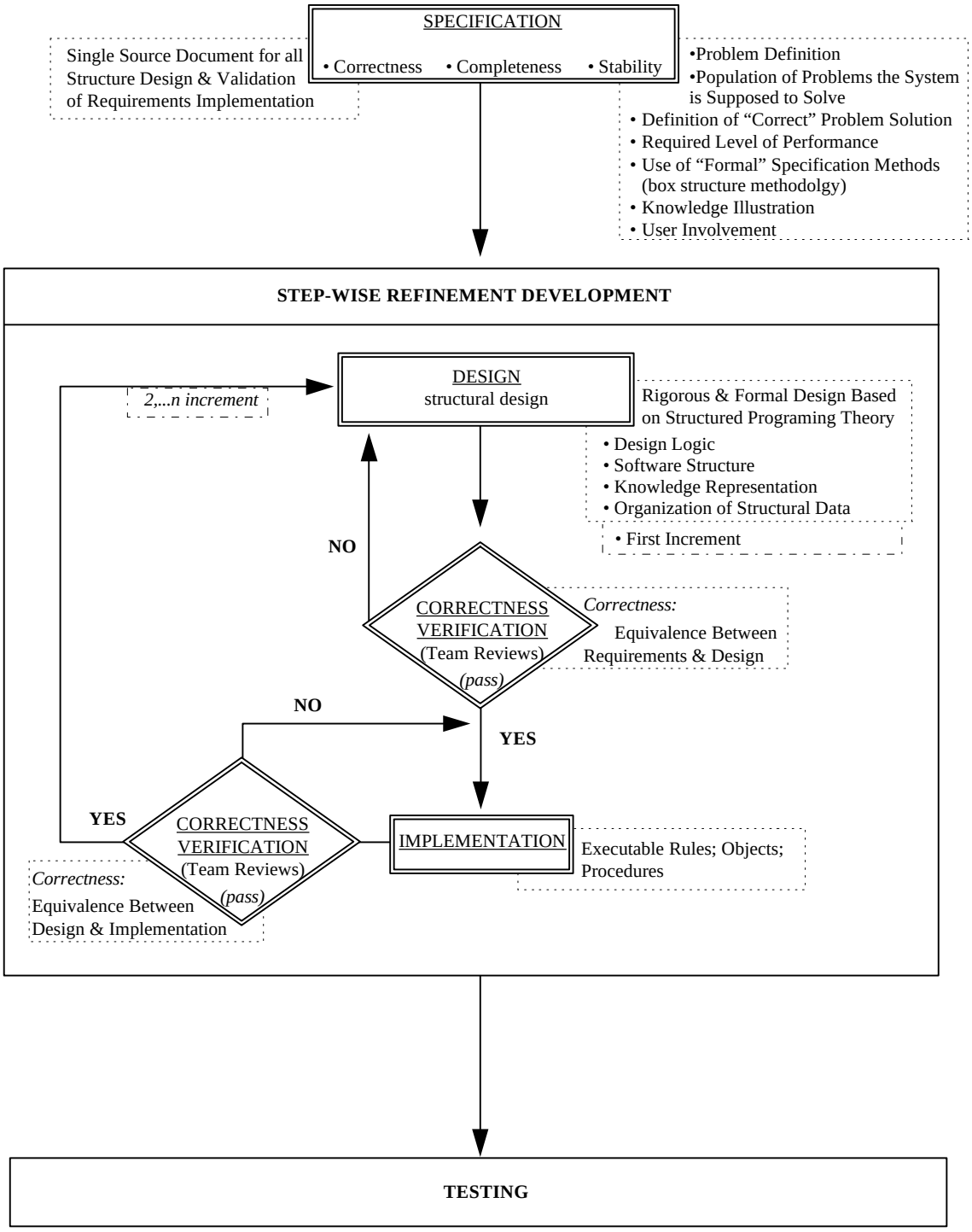

Figure 3.1: Developing a Verifiable System

Specification

The goal of this stage is to develop the system's specification.

input: software specific customer requirements.

output: software functional and performance requirements.

The Importance of Specifications

Specifications are important for VV&E. As noted in the introduction, *verification* determines if a system meets its specifications; this is meaningless if there are no specifications. *Validation* determines if a system does what is needed; this is only possible if it has been decided what a system is supposed to do. The results of these decisions are *specifications*.

At the specification stage the emphasis is on producing a clear identification of:

- What is to be produced?
- When to produce it?
- What are the resources required?

The issue is to find a trade-off between the requirements specification (client) and the resources (time and money). The use of formal approaches (formal notation i.e., the Structured Analysis [SA; De Marco 1978], the Software Requirements Engineering Methodology [SREM; Alford 1978], the Structured Analysis and Design Technique [SADT; a trademark of SofTech],) proved to be very useful in this process. This is especially important to the V&V task because of the clarification provided by the use of these methods.

Functional Specification (FS): Specification of functions to be performed by the system and the constraints within which it must work.

Acceptance Test Specification: Test definition:

- Who will perform the test(s)?
- When (at what point)?
- How do we insure that the system behaves according to the FS ?
- Include V&V Techniques to be used and when (at which time).

In addition to the above mentioned items, the following items should also be addressed:

- A clear definition of the population of problems the expert system is supposed to solve.
- A provision of test and development samples.
- The required level of performance.
- A clear definition of what constitutes a correct problem solution verification:
    - Is it possible to collect inputs that could possibly solve the problem?
    - Is it possible to compute the proposed output from the input validation?

- Can the experts certify that the specifications, if properly implemented, would solve the evaluation problem?
  - Can experts judge that the system is worth the probable cost?
  - Can experts judge that the system would be useful in practice?
  - Is it possible to build a system that could be integrated with other components as necessary?

The General Form of Specifications

For the formal proof techniques presented in the following chapters, it is useful to have a general representation of a specification. Most specifications are based on the following form:

> For some subset S of the input space of an expert system, and
> for all X in S,
> the output of the system satisfies some proposition P.

Defining Specifications

It is particularly important to define specifications for the critical cases the expert system may encounter. A *critical case* for an expert system is a set or range of input data on which failure of the expert system to perform correctly causes an unacceptable, perhaps catastrophic, failure of the system of which the expert system is a part.

There are several steps in defining and verifying specifications for an expert system:

- Gather informal requirements from experts, with particular attention to defining the critical cases.
- Obtain expert certification of the specifications.
- Validate informal descriptions of the specifications with experts.
- Validate the translation of informal specifications into the formal notation used in the knowledge base.
- Validate the formal statement of the requirements using symbolic evaluation.

Each step is detailed in a section below, with particular attention to critical cases.

Gather Informal Descriptions of Specifications

The first step in verifying specifications is to gather a complete set of requirements. Only the domain expert(s) can provide this list. Ideally, during the original knowledge acquisition phase for the expert system, the knowledge engineer gathered, documented, and validated the critical cases. If the informally stated requirements are not available, however, gathering them is the first necessary step in verifying the correctness of an expert system.

Typically, to gather the critical cases, the knowledge engineer should ask the domain expert(s) to list critical cases, and to keep a careful record of them. As with most knowledge acquisition tasks, it is important to ask for the following information:

- General principles, e.g. "What are the critical performance requirements for this expert system?"
- Specific projects, and the critical performance requirements found in those projects. To get this information, the knowledge engineer should ask the expert(s) to tell him about their projects and experiences that are within the scope of the knowledge base. The purpose of this is that by reviewing the specific projects the expert's memory will be spur. This process will help the engineer to decide what the critical cases really are.

In gathering a set of critical cases, it is important to let the domain experts describe critical cases in their own words and notation, not in the notation of the expert system. This is because the expert system may have missed a critical variable that may be needed to recognize a critical case. If the knowledge engineer asks the expert to verify knowledge base gobbledygook, the expert may become too distracted to think of a critical case not described with the incomplete set of variables used in the incomplete knowledge base.

Obtain Expert Certification of the Specifications

It is important that the knowledge engineer impel the expert(s) to certify the specifications, especially those concerning the critical cases. This is a vital step in the process because the expert system will be built to meet and tested against the specifications. If the specifications are in error, the expert system will almost surely fail to perform properly.

In order to obtain meaningful certification of the specifications, the knowledge engineer must make sure that the expert focuses on a careful review of the specifications. Among the ways to obtain this focus are:

- Have a group of experts reach consensus on the specifications, with the knowledge engineer functioning as a moderator. In this role, the engineer will:
    - Be familiar with the ongoing discussion, and in addition, will be in a position to solicit important issues that must be resolved.
    - Ensure that the experts address those issues and reach an agreement.
- Have the expert(s) sign off on the specifications.

Validating Informal Descriptions of Specifications

For systems where correct performance is critical, the next step in validating specifications of the expert system is to validate the informal descriptions of critical cases. The basic method for validation is that of cultural consensus, described in the chapter, "Validating Expert Knowledge." In this method, experts, ideally those who have not provided the specifications, are used to validate the correctness of those specifications.

There are two questions that should be asked concerning the informal list of critical cases to validate: is the set of critical cases complete, and are the critical cases correct? To validate completeness, the knowledge engineer should conduct interviews with experts who have not contributed to the critical case list. This interview is similar to the one used to gather the list of critical cases, with one additional step: at the end of the interview, ask the expert to certify not just the critical cases the expert proposed, but the entire list of critical cases gathered so far, including those that were added during the

interview. After additional experts no longer provide new critical cases, the entire list gathered has been validated to a confidence level depending on the number of experts who certify the list. Chapter 9, "Validating Underlying Knowledge", discusses these confidence levels in more detail.

Validating the Translation of Informal Descriptions

To validate the critical cases, the informal descriptions must be translated into formal statements in the language of the knowledge base. The goal of this translation is to produce statements of the form:

> if H1 and H2 ... and Hn then C1 and C2...and Cn.

The H's should be stated in terms of input variables of the expert system, and the C's should be possible conclusions of the expert system.

The translation into a knowledge base language is a process that can introduce errors. For example, for Knowledge Base 1 a critical case in the informal language of an expert might be, "If the client doesn't have a lot of money, he/she should first build a savings account." The closest that one can come to expressing this in the language of Knowledge Base 1 is:

> If "Discretionary income exists" = no
> then investment = "bank account".

A financial planner would probably consider "Discretionary income exists" an inadequate translation of "the client doesn't have a lot of money"; Knowledge Base 1 does not even ask about existing savings or most other assets.

As this example illustrates, the translation of expert knowledge into the formal knowledge language of an expert system is one of the tasks where errors can creep into the expert system. To have a truly validated expert system, the translation has to be validated. Although this is rarely done, items can be created for validation as follows:

- Is <expert's statement of a critical case>
- equivalent to <the same critical case in the knowledge language>

These items form the basis for a cultural consensus test for a set of knowledge engineers (see chapter 9 "Validating Underlying Knowledge"). When asking knowledge engineers to validate the translation of critical cases, it is important to:

- Use knowledge engineers who have not built the knowledge base.
- Give the validating knowledge engineers the opportunity to familiarize themselves with the knowledge language before examining the individual items.

In translating the informal requirements into formal knowledge base statements, there are some typical kinds of errors, as discussed below:

- False negatives in the input variables: One problem in knowledge translation results from the fact that a symptom is often used in a knowledge base to stand for an underlying condition; in the above example, for example, "no discretionary income" stands for "has no money." However, few observations are 100 percent reliable. If a single symptom is used to test for a condition in a knowledge base, a false negative of that symptom will produce an error in what the expert system does.

The solution to the false negative problem is to separate symptoms and underlying conditions in the knowledge base. If C is a condition, the knowledge base should contain a rule of the form:

if S1 or S2 or ... Sn then C (Rule C).

Where S1 through Sn are a set of symptoms such that the probability of false negatives in all the S's Is less than some agreed-on threshold. Outside of Rule C, and similar condition-inferring rules, the S's should not appear when a condition (i.e., C) is intended. Therefore, every occurrence of an S outside of a condition-inferring rule should be validated by expert(s).

In the case where a single symptom has such low false negatives that it identifies C by itself below the acceptable error threshold, it is unnecessary to separate the symptom and condition in the knowledge base:

- Missing input variables: An expert learns to observe many symptoms of possible problems. An expert system may use only a small number of variables. Whether the small number of variables is adequate is a matter that experts must validate. It is important to ask experts what data they gather in looking at problems covered by the knowledge base. If the expert looks at more than the expert system, for example variable X, then:

  Can the expert get along without <variable X>

  is a knowledge item that should be validated (see chapter 9).

Validation of Formalized Requirements

At this point, the critical cases have been transformed into a set of statements of the form:

if H1 and H2 and ... and Hn then C1 and ... Cm (name: f1).

Formal verification methods for specifications in this form are discussed in the chapters on knowledge modeling and verification techniques for small systems.

Figure 3.2 outlines the steps to be considered at the specification stage and figure 3.2.1 shows their implementation to knowledge base 1.

Other Issues to be addressed at this stage:

- Project Plan: Breakdown of the work; manpower figures, milestones, ect.
- Quality Management Plan: Quality Control.

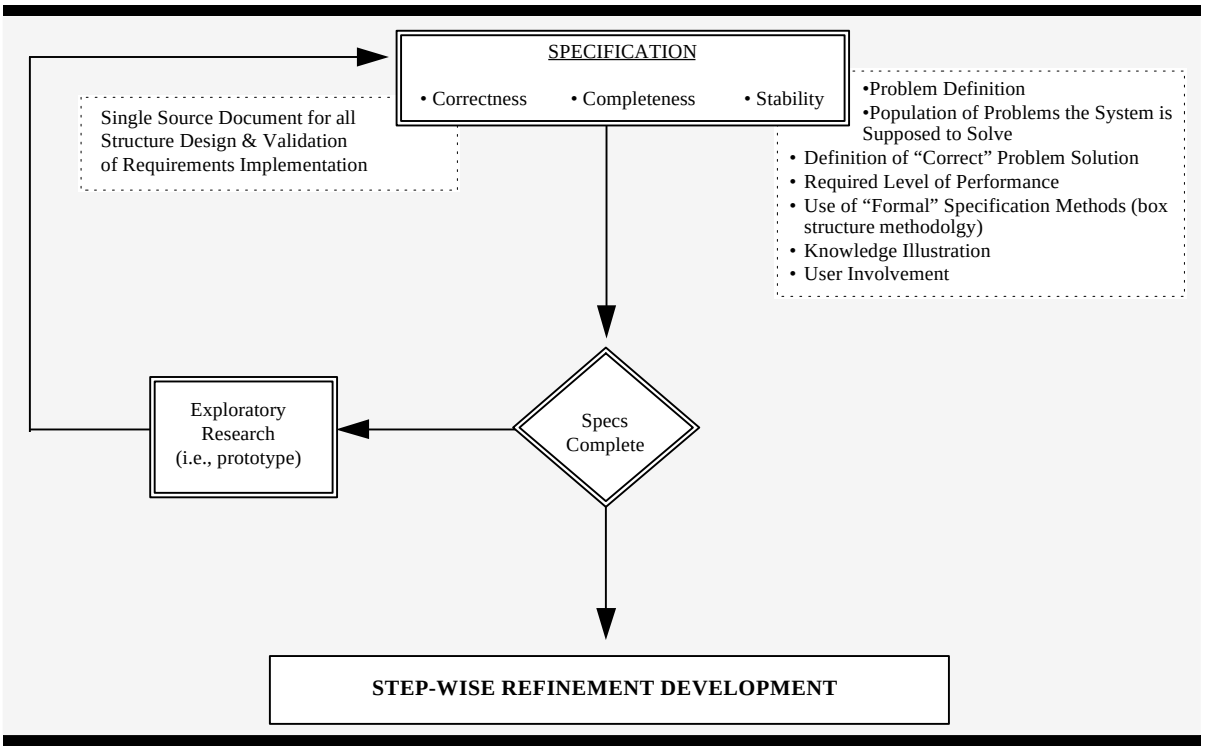

Figure 3.2: Specification

Problem Definition:

- The lack of readily available investment advice.
- Develop a system that will advise the user on investment strategies.

Population of Problems the System is supposed to solve:

- Investment advice to people with less than $ 1 Million to invest.

Definition of Correct Problem Solution:

- An investment strategy is always suggested.
- Proposed solution should be affordable
- The investor is comfortable with the advise.

Required level of performance:

- As good as 70% of the expert(s) [Define: good 70% of the time or 70% as good all the time].
- The system should always recommend an affordable investment even if it has to be a conservative one.

Note: Knowledge Acquisition & User Involvement:

Figure 3.2.1: KB1 Specification

Step-Wise Refinement Development

At this stage, a mapping of the system functions (from FS) into software components will occur and the overall System Structure (Architecture) must be defined. The use of the following Box Structure Methodology will help in this process.

Software Structure

The general software architecture should consist of:

- Software Components (for each software component, determine its purpose, functionality, interface, and data requirements).
- Structure & Flow.

Box Structure Methodology

- Black Box: External view of the system. This provides a system description of the user visible system inputs and responses. No details on the internal structure and operations are provided.
- State Machine: Intermediate system view. This decomposes the internal state structure from the BB description of the system.
- Clear Box: Internal view of the system operations on inputs and internal state data.

If the box structure methodology, is to be used, the first level/increment should consist of the overall design taken as deep as possible (using black boxes for functions and sub-systems). At every

subsequent increment the design should be taken deeper, two to three level down, until all the boxes are replaced by their respective functions/subsystems.

Design Refinement

The Top Level Design:

Given the specifications for the system as a whole, a top level software module is designed with the following properties:

- The design for the top level software is written in a language, which may be but is usually not a compilable programming language. Any language which has a precisely enough defined syntax and semantics to unambiguously define what the design does when executed, and to carry out required correctness proofs of the software can be used. Languages tha fit these requirements are called design languages. The process of rendering axiomized software into a design language is called designing the software.

- The software design can be translated from its existing language into a compilable programming language. Techniques for doing this translation for standard knowledge models will be presented later. The software design can be proved correct. In particular, the software can be proved to be complete, consistent and to satisfy its specifications, under the assumption that any other functions or other software modules used within the current object of proof satisfy some written, precise, mathematical specifications.

Refining the Design:

Once the disign process has been started, a modification of the familiar successive refinement lifecycle adds detail to the design. Detail is added in two ways:

- Software modules which have been axiomized but not designed can be designed.

- Software that has been designed can be translated into a language that is closer to, or is, a compilable programming language.

Verifying the Design:

A design is verified when it has been proved that the designed module is complete, consistent and satisfied its specifications.

- A module is complete iff for all points in its input space, some values of the outputs and behaviors required to instantiate the specifications are computed.

- A module is consistent iff it is possible, both mathematically and under the constraints imposed by knowledge in the area of application, for all the output values and behaviors to be true at the same time.

- A module satisfies its specifications iff its specifications are true when instantiated with any input values and any outputs or behaviors produced from those inputs.

Completing the Design:

A design is complete when all software modules appearing in the design have been axiomized, designed and verified.

From Specifications:
1. An Investment strategy is always suggested:
List of possible investments strategy for KB1:
- Stocks.
- Saving Accounts.

Do Nothing (Although this is a good choice for many instances, it is not considered for the example).
Note: The list of output might be incomplete at this stage (i.e., may discover other possible strategies down the line).
Define the specifications in terms of these newly defined list of output.
2. Proposed solution should be affordable:
- When is stock affordable?
- When is Saving Account affordable?

Interaction with the expert(s)
Depending on the complex nature of the questions to be answered, we may find out that other things might be needed:
- Interaction with data bases.
- Algorithmic routines.
- Sub Expert systems.

For KB1:
The expert determined that stocks are affordable if "Discretionary Income" exists.
We have to define "Discretionary Income" in a measurable manner.
From the interaction with the expert, we introduce the concept that in order to have "DI", the investor has to have:

Some savings (> $ 3000.).
A luxury item (Boat/ Luxury Car).

n.b.: 1. Keep careful records of interaction with the expert(s).
2. One of the products of these steps are expert(s) verifiable statements about the knowledge domain.

i.e., Stocks are affordable if there is savings and a luxury item.
These will be used for carrying out formal proof procedures. In a high risks situation (see table 3.1) these statements should be verified by enough experts to get the required level of confidence (see chapter 9).
We have preliminary design information that consists of:
1. An expert sub-system to determine affordability.
2. An expert sub-system for risk tolerance.

---

3. An expert sub-system which makes an investment category decision using 1 & 2.
n.b.: This is very useful for designing a well structured system.
Refer to chapter 7, "Knowledge Modeling", and pick a knowledge model that fits the preliminary design information.

Figure 3.2.2: KB1 Design

Implementation

In the implementation step, a software module is translated from its current design language into a compilable language. The source code resulting from that translation must be verified, i.e. shown to be complete, consistent and to satisfy its specifications.

Implementation is the last step in a series of design and translation steps that turn an initial high level specification for the system as a whole into compilable code. The stepwise refinement process that produces the code from the initial specification uses the following refinement operations:

- Design of a module that has been specified but not yet designed.
- Specification of a module that is used in a designed module but has not yet been specified.
- Translation of a module from one design language to another language, usually one that is more detailed and closer to a compilable language.

At the Implementation stage, the main objective is the creation of a complete executable system, including software to carry out all processes specified in clear or black boxes, according to constraints on those parts of the system. The system is comprised of executable rules, objects, procedures, etc., that:

- Satisfy requirements of the system as a whole .
- Are executable functions that are equivalent to abstract functions specified in the design.

For example, the design may specify a function that determines that the user is rich. The implementation may check the bank account, kind of car owned, etc. However, it may not catch certain rich people because it does not check art owned. In this case, the implementation fails to carry out the abstract function required of it. In general, the computer bases a conclusion on less observed data than an expert, and simplifies the inference an expert makes to one that is just based on the small set of data the computer looks at.

The implementation stage should consist of the following steps:

1. Determine the high level structure of the system to be implemented.
2. Define communication between subsystems Implementation.
3. Provide a detailed definition of subsystems.

4. Select the implementation tool.

5. Execute the implementation in the tool.

Constraints on Design and Implementation

The following constraints apply to the operations in the stepwise refinement process:

Specification of a module must include all properties that are used in any existing verifications of other modules.

- No module can be designed before it is specified.
- Designs must be proved to satisfy their specifications.
- Translations must preserve specified properties of the source module (being translated) in the destination module (the result of the translation).

Correctness Verification

Design vs. Specification

The overall result of this is a proof that any system that satisfies all the design documents is correct (i.e., complete, consistent, stable, satisfies requirements imposed by subject) provided that the parts not yet designed or implemented have properties as required by clear box theorems and the models of knowledge, or specified by the expert.

Code vs. Design

The equivalence between requirements and implementation must be proven. Previous results together with proof of equivalence of design and implementation may be used. This may take the form of a cleanroom-type layered correctness proof in which all boxes are clear and implemented, with the top part constituting the previous proof of the equivalence of requirements and design.

Depending on the complexity of the problem and the consequence of failure, this process is to be accomplished by the developer(s) (Level I), the developer(s) and two members of the organization (Level II), or a separate verification team (level III). Table 3.1 is to be used as a guide in determining the level of the project. figure 4.3 shows the process and figure 3.3.1 is the implementation to knowledge base 1.

Table 3.1: Level of Effort for the Correctness Verification Stage

	Consequence of Failure				
Complexity	**Loss of Life**	**Injury**	**High $$$**	**Inconvenience**	**Other(lC,...)**
Very Complex	III	III	III	II	I
Medium	III	III	II	I	I
Simple	III	III	II	I	I

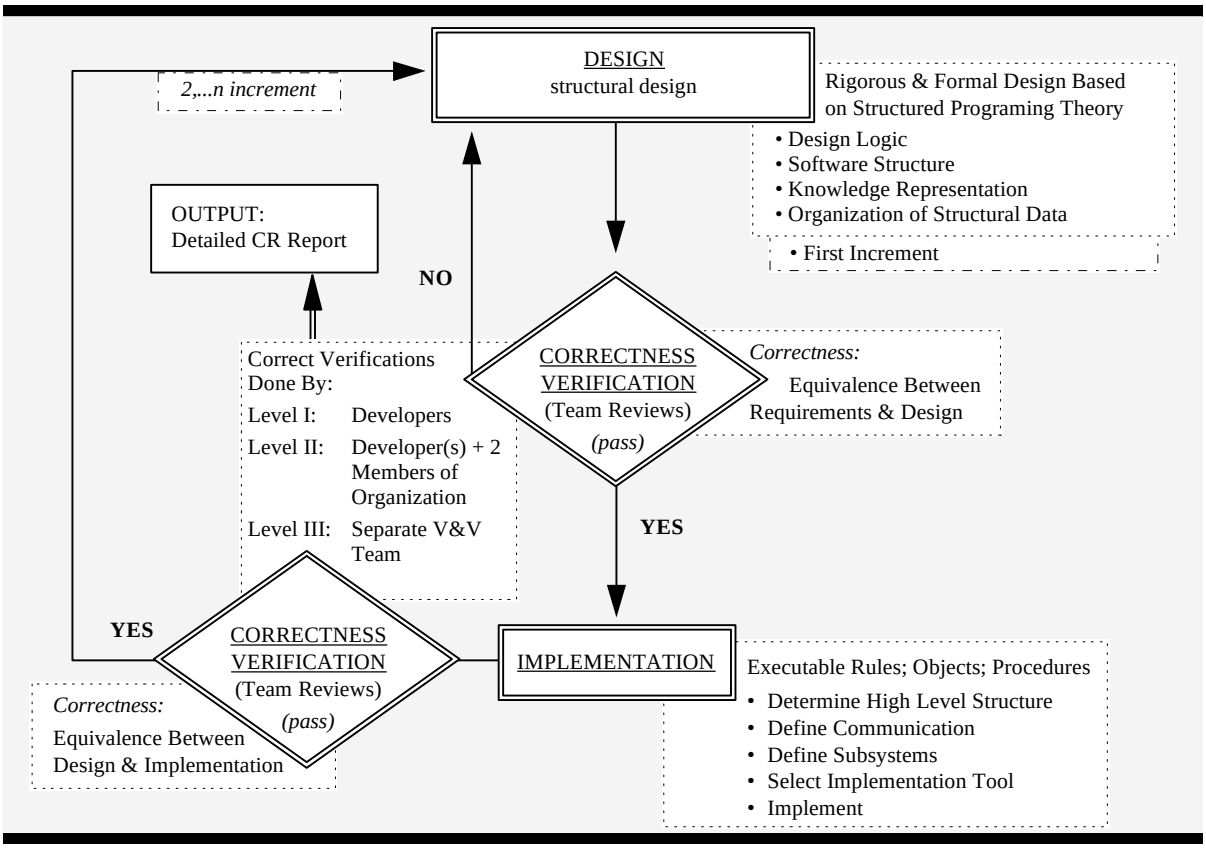

Figure 3.3: Correctness Verification

Step 1 -- Determine the high level structure of the system to be implemented: From the design stage, it was determined that the expert system consists of 3 subsystems, discretionary income (DI),

risk tolerance (RT) and type of investment (INV). The structure of the system can be expressed by the function:

Investment = INV(DI(boat, "luxury car", "savings account"),

RT(stocks, "lottery tickets"))

This expresses the fact that the output of DI and RT are inputs to INV.

Step 2 -- Define communication between subsystems: The output of DI and RT must be sufficiently fine-grained to distinguish cases where different investments are indicated. Since there are only 2 investments in this example system, only 2 values are required as output for each of these subsystems; use *high* and *low* for risk tolerance, and *yes* and *no* for discretionary income. At this point, the inputs, outputs and communication between subsystems have all been defined.

Step 3 -- Detailed definition of subsystems: In this stage, the expert information collected in the design step will be converted into precise logical statements; this process will be illustrated on the DI subsystem.

The condition that must be true to have discretionary income is:

A = (Savings > $3000) (1)

AND ("Own Boat" = yes OR "Own Car" = yes)

The expert information about discretionary income can be formalized as:

A IMPLIES ("discretionary income" = yes) (2)

NOT A IMPLIES ("discretionary income" = yes) (3)

Step 4 -- Selection of implementation tool: At this point, there is enough information to choose a tool in which to implement the expert system. The requirements on the tool are:

- Provide for communication between subsystems.
- Express rules such as (2) and (3).

Most rule-based expert system shells meet these requirements. Although the order of information in the knowledge base must be slightly different in forward and backward chaining implementations, either form of inference engine can be used to implement this knowledge base.

Step 5 -- Implementation in the tool: The rule-based-shell implementation will be written in two steps: first as a generic rule-based implementation, finally as an implementation in CLIPS.

Step 5.1 -- A generic rule-based implementation: Rule-based shells typically allow menu, fill-in and yes-no questions. The following questions will gather the necessary information for discretionary income:

QUESTION TEXT	TYPE
What is your savings balance?	fill-in
Do you own a boat	yes-no
Do you own a luxury car	yes-no

The inputs and outputs can be represented inside the expert system by the following variables:

VARIABLE	TYPE	VALUE
savings	numerical	>= 0
"Do you own a boat"	boolean	yes or no
"Do you own a luxury car"	boolean	yes or no
"discretionary income"	enumerated values	high or low

Now put the knowledge in statements (2) and (3) into the rule form of rule-based shells. Rule based shells encode information in the following form:

• Rules are of the form:

IF <conditions> then <inferences> and <actions>

• <conditions> are built from simple requirements with the logical operations AND, OR and NOT.

• Many of the simple requirements can be written in the forms such as VARIABLE = VALUE, or more generally:

VARIABLE REL VALUE, where REL is one of the relations

=, >, <, >=. <=

• Inferences can also be written in the form:

VARIABLE = VALUE, i.e. VARIABLE is set equal to VALUE.

Actions are dependent on particular shells, and will be deferred at this time.

Using the above notation, (2) can be written as:

IF (Savings > $3000) (4)

AND ("Do you own a boat" = yes

OR "Do you own a luxury car" = yes)

THEN "Discretionary income" = yes

(3) can be put into rule form as:

If NOT <if part of (4) THEN "Discretionary income" = no (5)

Alternatively and more usually, a rule implementing (3) is written in a form in which the NOT is applied individually to the simple requirements contained in the "IF" part, rather than to a complicated expression built up from requirements. *DeMorgan's Laws* in mathematical logic:

NOT (A OR B) = NOT A AND NOT B (6)

NOT (A AND B) = NOT A OR NOT B

Using (6) repeatedly transforms (5) to:

IF (NOT Savings > $3000) (7)

OR

(NOT "Do you own a boat" = yes

AND NOT "Do you own a luxury car" = yes)

THEN "Discretionary income" = no

Simplifying the simple conditions using the following relations,

(NOT Savings > $3000) = (Savings <= $3000) (8)

(NOT "Do you own a boat" = yes)

= ("Do you own a boat" = no)

(NOT "Do you own a luxury car" = yes)

= ("Do you own a luxury car" = no)

Substituting (8) into (7) gives:

IF (Savings <= $3000) (9)

OR ("Do you own a boat" = no

AND "Do you own a luxury car" = no)

THEN "Discretionary income" = no

Figure 5.1 shows an expert system in a generic rule-based shell language that implements the discretionary income, risk tolerance and investment subsystems. The result is a small knowledge base (called Knowledge Base 1) that implements the investment expert system. [Note: Knowledge Base 1 leaves out the savings requirement, to further simplify the example when it is used to illustrate verification and validation.]

Step 5.2 -- Implementation in CLIPS

Once a generic knowledge base has been written, it must be translated into the language of a particular shell. Shown below is an implementation of the generic knowledge base in CLIPS. The CLIPS is fairly close to the generic rule-based KB. The main differences are:

rule syntax: Rules in CLIPS have the following syntax:

```
(defrule <RULE NAME> <COMMENT>
 <LIST OF CONDITIONS>
 =>
 <LIST OF ACTIONS AND INFERENCES>
)
```

implementation of the AND operation: The AND operation can be implemented in two ways:

- A list of the conjuncts in the AND.
- An explicit AND operation.

These alternative ways of writing AND are illustrated by the following two equivalent rules:

```
(defrule rule1  "stock"
 (risk_tolerance high)
 (discretionary_income TRUE)
 =>
 (assert (investment stocks))
 (printout t "We recommend stocks." crlf)
)
(defrule rule1  "stock"
 (and (risk_tolerance high) (discretionary_income TRUE))
 =>
 (assert (investment stocks))
 (printout t "We recommend stocks." crlf)
```

```
)
```
implementation of the OR operation: The OR operation can be implemented by an explicit OR operation, i.e.,
```
(defrule rule3c  "high risk tolerance"
 (or  (now_own_stocks TRUE )(lottery_tickets TRUE ))
 =>
 (assert(risk_tolerance high))
)
```
Equivalently, one can write a separate rule for each disjunct in the OR:
```
(defrule rule3c1  "high risk tolerance 1"
 (now_own_stocks TRUE )
 =>
 (assert(risk_tolerance high))
)
(defrule rule3c2  "high risk tolerance 2"
 (lottery_tickets TRUE )
 =>
 (assert(risk_tolerance high))
)
```
Here is an actual CLIPS implementation. This implementation is a fairly straightforward translation of the generic KB1. More sophisticated implementations of KB1 would structure the knowledge base so that when sufficient information for a conclusion had occurred, the user would be spared extra questions.
```
;
; KB1 in CLIPS,  a demo rule based system
;
;
; Note: In the following knowledge base,
; we will use certain user interface functions
; which can be defined in CLIPS:
;
; yes-or-no-p asks a yes-no question
; ask-parm asks a fill-in question
; ask-parm asks a menu question
;
; To run this CLIPS knowledge base, you need these functions
; which are not shown here.
;
;
; INVESTMENT TYPE SUBSYSTEM
;
;    Rule 1: If "Risk tolerance" = high
;      AND "Discretionary income exists" = yes
;         then investment = stocks.
;
(defrule rule1  "stock"
 (risk_tolerance high)
 (discretionary_income TRUE)
```

```
 =>
 (assert (investment stocks))
 (printout t "We recommend stocks." crlf)
)
;
;   Rule 2: If "Risk tolerance" = low
;     OR "Discretionary income exists" = no
;        then investment = savings account.
;
(defrule rule2a  "savings account 1"
 (risk_tolerance low)
 =>
 (assert (investment "savings account"))
 (printout t "We recommend a savings account." crlf) )

(defrule rule2b  "savings account 2"
 (discretionary_income FALSE)
 =>
 (assert (investment "savings account"))
 (printout t "We recommend a savings account." crlf) )
;
; DISCRETIONARY INCOME SUBSYSTEM
;
;   Rule 5: If
;        ( Savings > $3000)
;        AND ("Do you own a boat" = yes
;          OR "Do you own a luxury car" = yes)
;        then "Discretionary income exists" = yes.
;
; First we will gather the information
;
(defrule rule5a  "boat"
 (not (has_boat ? ))
 =>
 (bind ?boat ( yes-or-no-p "Do you own a boat? " ))
 (assert (has_boat ?boat  ))
)
;
(defrule rule5b  "luxury car"
 (not (has_lux_car ? ))
 =>
 (bind ?lc ( yes-or-no-p "Do you own a luxury car? " ))
 (assert (has_lux_car ?lc  ))
)
;
(defrule rule5c  "savings balance"
 (not (savings_balance ? ))
 =>
 (bind ?sb ( ask-parm "What is your savings balance? " ))
 (assert (savings_balance ?sb))
)
;
; Now we will use the information determining discretionary income
```

```
;
(defrule rule5d  "has discretionary income"
 (savings_balance ?sb)
 (test( > ?sb 3000))
 (or (has_lux_car TRUE  ) (has_boat TRUE))
 =>
 (assert (discretionary_income TRUE))
)
;
;    Rule 6: If Savings <= $3000
;              OR
;              ("Do you own a boat" = no
;                AND "Do you own a luxury car" = no)
;              then "Discretionary income exists" = no.
;
(defrule rule6  "has no discretionary income"
 (savings_balance ?sb)
 (test( <= ?sb 3000))
 (and (has_lux_car FALSE  ) (has_boat FALSE))
 =>
 (assert (discretionary_income FALSE))
)
;
; RISK TOLERANCE SUBSYSTEM
;
;    Rule 3: If "Do you buy lottery tickets" = yes
;      OR "Do you currently own stocks" = yes
;         then "Risk tolerance" = high.
;
(defrule rule3a  "lottery tickets"
 (not (lottery_tickets ? ))
 =>
 (bind ?Lt ( yes-or-no-p "Do you purchase lottery tickets ? "))
 (assert (lottery_tickets ?Lt ))
)
;
(defrule rule3b  "currently own stocks"
 (not (now_own_stocks ? ))
 =>
 (bind ?s ( yes-or-no-p "Do you currently own stocks ? "))
 (assert (now_own_stocks ?s ))
)
;
(defrule rule3c  "high risk tolerance"
 (or  (now_own_stocks TRUE )(lottery_tickets TRUE ))
 =>
 (assert(risk_tolerance high))
)
;
;    Rule 4: If "Do you buy lottery tickets" = no
;      AND "Do you currently own stocks" = no
;         then "Risk tolerance" = low.
;
```

```
(defrule rule4  "low risk tolerance"
 (and (now_own_stocks FALSE)(lottery_tickets FALSE))
 =>
 (assert(risk_tolerance low ))
)
```

Figure 3.3.1: KB1 Implementation